An Extension to PDDL: Actions with Embedded Code Calls

Okhtay Ilghami
University of Maryland at College Park
okhtay@cs.umd.edu

J. William Murdock
IBM Watson Research Center
murdockj@watson.ibm.com

Abstract

In most existing planning systems, *plan generation* and *plan execution* are two entirely separate phases. This fact has had a huge effect on the way PDDL has been developed so far: It is assumed that the two aforementioned phases are separable. In this paper, we investigate the characteristics of the domains in which this separation is impossible. We also propose extensions to PDDL, both syntactically and semantically, which will make PDDL capable of describing such domains.

Introduction

In traditional planning systems, *plan generation* and *plan execution* are separated. This simplifying separation is based on the usually unrealistic assumption of *perfect knowledge*. In reality, however, there can be two reasons for the lack of such knowledge: The state of the world may not be fully known, or actions may have unpredictable effects. In either case, the planner may need to execute some of the steps of the generated plan to observe more about the state of the world and/or the preconditions and effects of the actions in those steps. The consequence of executing actions while generating plans is that the state of the world is changing and these changes are irreversible. For example, consider a robot trying to get out of a maze of unknown configuration. The robot may initially move around to determine where the walls are, and then start planning using that information.

PDDL (Fox & Long 2001b), a purely descriptive and platform-independent language, has recently become the de facto standard for defining planning problems and evaluating plans. Perfect knowledge is assumed in the existing PDDL specification. In this paper, we propose a way to extend PDDL so that it can be used to describe domains where the planner does not necessarily have perfect knowledge, and it may execute code calls within the actions to gain knowledge that may be useful in generating later steps of the plan. In the next section, we discuss the syntactic and semantic aspects of such an extension, and how it affects the notion of a valid plan. We then conclude the paper by sections on related work and future directions.

Adding Actions with Code Calls to PDDL

In this section, we explain how our extension to PDDL is formulated, and how it affects the semantics of a domain and a plan. We propose three extensions to the PDDL syntax:

i) Two new possible requirements for a PDDL domain: `:code-call-actions` to declare that a domain contains actions with code calls, and `:code-call-durative-actions` to declare that a domain contains durative actions with code calls.

ii) A new kind of instantaneous action, denoted by the keyword `:code-call-action`: These actions look exactly like ordinary PDDL actions, except for one extra construct, denoted by `:code` in their definition. This construct consists of a predicate analogous to a function call, and a list of typed returned values. The names of these variables must start with $\#$. These variables can be used in the effects of the action, as any other variable.

iii) A new kind of durative action, denoted by the keyword `:code-call-durative-action`: These actions look exactly like PDDL durative actions, except for one extra construct, denoted by `:code` in their definition. This construct can be temporally annotated, and it consists of a predicate analogous to a function call, and a list of typed returned values. The names of these variables must start with $\#$, and they can be used in the effects of the action.

In this paper we discuss only instantaneous actions with embedded code calls. It is straightforward to generalize our discussions both syntactically and semantically to include durative actions with embedded code calls. From now on we use the term *action* to refer only to *instantaneous actions*.

Consider the k-armed bandit problem (Berry & Fristodt 1985; Kaelbling, Littman, & Moore 1996). In this problem, there are k gambling machines and a robot. When the robot pulls the arm of the ith machine, the machine pays off either a dollar with the constant probability p_i or nothing with the probability $1–p_i$. The robot is allowed to have a fixed number of pulls, h, and p_is are not known to the robot in advance. The goal is to maximize the total pay off. Since the robot does not know the p_is, it cannot simply plan and execute separately. The robot can gain information (i.e., get a better approximation of p_i) by pulling the arm of the ith machine. This is an irreversible action, since it decreases the number of allowed pulls by one. In Figure 1, we show how this problem can be defined in our proposed extension of PDDL. Action `pull` uses a code call `do-pull` to pull a lever and see what happens. This code call is defined in the line tagged `:code`. The code call returns a single numeric

```
(define (domain K-armed-bandit)
 (:requirements :fluents :adl
  :code-call-actions)
 (:types lever number)
 (:functions (pulls) (cash))
 (:code-call-action pull
  :parameters (?l - lever)
  :precondition (> (pulls) 0)
  :code ((do-pull ?l) (#pay - number))
  :effect (and (decrease (pulls) 1)
               (increase (cash) #pay))))
```

Figure 1: Encoding the K-armed bandit problem

value. This value is used to instantiate variable `#pay`. Although this code call returns only one numeric value, code calls can return several return values with arbitrary types.

In Figure 2 we show how to define the problem of a robot trying to get out of a maze. In this problem, the robot has an initial position and wants to get to a final position. It has four possible actions to move north, south, east or west. The robot cannot plan in advance and execute later since it does not know where the walls are located. The only way the robot has to figure out if there is a wall to its north is to try to move north. The return value `#ret` indicates whether the robot has moved north or has been stopped by a wall.

Actions with embedded code-calls are useful to describe actions with these three characteristics: First, the outcome of an action is not known in advance. This can happen for several different reasons, for example imperfect knowledge about the current state of the world. Since the effects of the action are not known fully in advance, it cannot be defined as a classical planning action. Second, although the outcome of the action is not known in advance, the planner may speculate that executing it may help to reach the goal (either directly or indirectly via the information gained by executing the action and then observing the results). Third, executing the action changes the state of the world.

Embedded code calls are particularly useful for sensing actions (i.e., actions that give the planner some information about the outside world). In real world, sensing actions can be performed by some external agent. For example, the planner may control a robot that can measure the temperature, navigate the outside world, etc. We define an *oracle* to be an abstraction of this external agent in our framework. Anytime the planner may wish to actually execute something, it does so by sending a message to the oracle. We assume that the oracle executes the action and returns the results of doing so back to the planner. In our proposed extension, each planning domain is coupled with an oracle O, which is responsible for processing code calls.

PDDL is a purely descriptive and implementation-independent language. Our code call syntax preserves these traits; our syntax is a general format for planners to communicate with oracles (i.e., the external agents). A code call is a general form of a function call. It is a predicate with two elements. The first element is analogous to a function call (function name followed by its arguments). The second element is a list analogous to the return values of the function

```
(define (domain robot-and-maze)
 (:requirements :fluents :adl
  :code-call-actions)
 (:types number)
 (:functions (x) (y))
 (:code-call-action north
  :parameters () :precondition ()
  :code  ((move (x) (+ (y) 1))
                (#ret - number))
  :effect (when (= #ret 1)
                (increase (y) 1)))
 (:code-call-action south
  :parameters () :precondition ()
  :code  ((move (x) (- (y) 1))
                (#ret - number))
  :effect (when (= #ret 1)
                (decrease (y) 1)))
 (:code-call-action east
  :parameters () :precondition ()
  :code  ((move (+ (x) 1) (y))
                (#ret - number))
  :effect (when (= #ret 1)
                (increase (x) 1)))
 (:code-call-action west
  :parameters () :precondition ()
  :code  ((move (- (x) 1) (y))
                (#ret - number))
  :effect (when (= #ret 1)
                (decrease (x) 1))))
```

Figure 2: Encoding the robot and maze problem

call. We assume whenever a planner decides to put an action with an embedded code call in a plan, the appropriate code call is executed, and the oracle instantiates the variables in the return value list. This instantiation is an abstraction of what happens in the real world: The planning system asks an outside agent for some information, and then continues planning using the information provided by that agent. The effects of asking the external agent to do so can be mentioned in the effects part of the action.

The details of how the actual oracle is implemented is outside the scope of PDDL. A Java implementation of the oracle may translate the code call to a member function of an object, while a Lisp implementation may translate the code call to a Lisp expression. From the point of view of someone using PDDL to define a domain, the underlying structure and implementation of the attached oracle must be transparent.

In our proposed extension, whenever an action with an embedded code call is used in a plan, the return values of the code call must be listed in the generated plan, in the same order they are mentioned in the domain definition, after the action. For example, if the robot uses the action `north` in the problem defined in Figure 2 in order to try to go north and it fails, the corresponding action listed in the plan will look like `(north)[0]`, and if the move is successful it will look like `(north)[1]` rather than simply `(north)`.

Whenever there are actions with embedded code calls listed in a planning domain, satisfaction of the precondi-

tions and achieving all the goals are not the only measures of validity of a plan. Since the planner cannot backtrack on certain actions, once it decides to invoke them using the corresponding code calls to the oracle it *must* include them in the generated plan. Therefore, the validity of a plan is defined with respect to the oracle O in the planning domain: *A valid plan must include all the actions with embedded code calls corresponding to the code calls it made to the oracle and the return values it got, in the same order, in addition to the traditional conditions for validity*.

Related Work

One approach to operating in domains where the starting state is not completely known or there is uncertainty in the effects of some actions is to interleave reasoning about what actions to execute with the actual execution. For example, reinforcement learning techniques (Watkins & Dayan 1992; Kaelbling, Littman, & Moore 1996) select actions using a simple numerical policy and incrementally learn improved policies based on the rewards obtained from performing actions. Agent centered search techniques (Korf 1990; Koenig 2001) also interleave execution and learning; unlike reinforcement learning, these techniques also perform some planning/search. Reflection using functional process models (Stroulia & Goel 1995; Murdock & Goel 2003) also interleaves reasoning and action, using a variety of planning and learning algorithms. We feel that our proposed mechanism for code calls is potentially useful for all of these methods.

There are also a variety of approaches that deal with incomplete knowledge and uncertain actions without interleaving reasoning and execution. For example, SGP (Weld, Anderson, & Smith 1998) performs contingency planning (i.e., it produces plans that include sensing actions and restrictions on which actions are performed depending on the results of those sensing actions). Similarly, CGP (Smith & Weld 1998) performs conformant planning (i.e., it produces plans that accomplish the goal regardless of the outcome of any actions). In general, approaches that handle incomplete knowledge and uncertain effects without interleaving reasoning and acting have significant disadvantages. For example, contingency plans can be very large and time-consuming to construct, and conformant plans frequently do not exist or have low quality. However, in some domains these disadvantages are outweighed by the benefits of not having to execute any actions until all planning is complete.

Code calls are not essential for planning systems that do not interleave planning and acting. However, even for those planning systems, it may be useful to integrate information needed for planning with information needed for execution. Such an integration can be useful for executing plans after planning is complete. The ability to execute plans is not important for traditional planners, but is important for many real-world applications of planning (e.g., web services, robotics, interactive agents). To the extent that PDDL can be valuable for serving as a common domain language for these sorts of systems, the addition of a mechanism for code calls seems productive.

Future Directions

This paper is meant to be a first step toward extending PDDL to handle situations in which plan generation and plan execution cannot be separated. There are still many unanswered questions and interesting topics for future research:

Although oracles are well-defined abstract entities in theory, there are issues to be addressed while implementing them in practice. Some of the questions that should be answered are: What are the effects of different programming paradigms, such as structured programming, object-oriented programming, and functional programming on the process of implementing an oracle? Do these different paradigms, in practice, affect the planning process too? What are the conceptual and practical side-effects of the assumption that there is an oracle attached to each planning domain?

Another interesting topic is the characteristics that an actual planning system needs in order to be able to handle actions with embedded code calls. Does our proposed PDDL extension have any unexpected consequences when employed in such a system? If so, are there revisions that can address these consequences?

Another question to be answered is how to enhance the framework we provided here to support other extensions proposed for PDDL. For example, are there any conceptual or practical problems in adding actions with embedded code calls to, for example, PDDL+ (Fox & Long 2001a)?

References

Berry, D. A., and Fristodt, B. 1985. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall.

Fox, M., and Long, D. 2001a. PDDL+ level 5: An extension to PDDL2.1 for modelling planning domains with continuous time-dependent effects. Technical report, University of Durham, UK.

Fox, M., and Long, D. 2001b. PDDL2.1: An extension to PDDL for modelling time and metric resources. Technical report, University of Durham, UK.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. P. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.

Koenig, S. 2001. Agent-centered search. *Artificial Intelligence Magazine* 22(4):109–131.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2–3):189–211.

Murdock, J. W., and Goel, A. K. 2003. Localizing planning with functional process models. In *Proceedings of the 13th Int'l Conference on Automated Planning and Scheduling*.

Smith, D. E., and Weld, D. S. 1998. Conformant graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 889–896. AAAI Press.

Stroulia, E., and Goel, A. K. 1995. Functional representation and reasoning in reflective systems. *Journal of Applied Intelligence* 9(1):101–124.

Watkins, C. J. C. H., and Dayan, P. 1992. Technical note: Q-learning. *Machine Learning* 8(3).

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 897–904. AAAI Press.